An Interface to Support the Identification of Dynamic MPI 2 Processes for Scalable Parallel Debugging

Christopher Gotbrath[1], Brian Barrett[2], Bill Gropp[3], Ewing "Rusty" Lusk[3], and Jeff Squyres[4]

[1] Etnus, LLC, 24 Prime Park Way, Natick, MA 01760
Chris.Gottbrath@etnus.com
http://www.etnus.com
[2] 415 Lindley Hall, Computer Science Department, Bloomington, IN 47405
brbarret@osl.iu.edu
[3] Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439
{gropp,lusk}@mcs.anl.gov
[4] Cisco Systems, Inc., 225 East Tasman Dr., San Jose, CA 95134
jsquyres@cisco.com

Abstract. This paper proposes an interface that will allow MPI 2 dynamic programs – those using MPI SPAWN, CONNECT/ACCEPT, or JOIN – to provide information to parallel debuggers such as TotalView about the set of processes that constitute an individual application. The TotalView parallel debugger currently obtains information about the identity of processes directly from the MPI library using a widely accepted proctable interface. The existing interface does not support MPI 2 dynamic operations. The proposed interface supports MPI 2 dynamic operations, subset debugging, and helps the parallel debugger assign meaningful names to processes.

1 Introduction

MPI style parallel applications can comprise anywhere from one to many thousands of processes running on anything from a single user's workstation to the largest supercomputing clusters. Regardless of the scale of the application, when it fails to behave as expected and a developer sits down to debug it the first thing that they need to do is get their parallel debugger attached to their parallel program. This means that the debugger has to connect to not one but many processes running on both local and remote nodes. To the user this is a simple command or a simple 'click' in the interface of a parallel debugger like TotalView. The parallel debugger is able to fufill this request becuase the MPI library provides information about what processes running on both local and remote nodes constitute the parallel application.

This paper proposes a new interface between MPI processes and the TotalView parallel debugger that will enable users to debug applications taking advantage of the MPI 2 dynamic process capabilities to spawn new processes or combine two separately started parallel applications into one application. The dynamic nature of these applications provides a complex challenge to the debugger. Not only does the set of processes change over time but the performance focus of MPI leads vendors to favor highly asynchronous MPI library designs. The information that the debugger needs to get the debugging session started is something the MPI library is designed to keep distributed and balanced.

This new interface builds upon the foundation of and lessons learned from the current MPI -1 TotalView process acquisition interface [5] and the current MPI-1 TotalView message queue display interface[2] both of which have been almost universally adopted by MPI vendors over the past 9 years. To understand the new interface it helps to review the current process aquisition interface at a general level.

To attach to a parallel program TotalView first needs to attach to the starter process. This gives it the ability to halt and resume the process, set breakpoints and both read from and write to the program state. The debugger establishes – on the basis of these capabilities – an interface with the MPI library used by the application. If the debugger attaches to the starter before the starter program has created the parallel job the debugger runs the starter to the point that the parallel job exists but has not yet run user code. At this point the debugger reads a specified data structure in the starter process that holds a list of the MPI processes and information such as the network address of the node on which each process is running.

The user can then be prompted with a list of all the processes and can make an initial decision on which processes need to be actively debugged. In order to attach to the remote processes that the user selected TotalView needs a remote debugging agent, called a tvdsvr, on each node that hosts one or more selected processes. These are started by the debugger itself, often using ssh. The tvdsvr processes attach to each of the selected MPI processes and communicate back over the network to the debugger. At this point the user is attached to their parallel job. Any processes that were not chosen for debugging are now released to start running user code. The developer can now examine and control the selected set of processes. At any point during the debugging session the user can change the selected set of processes, choosing to look at more or less of the ongoing parallel job.

This interface, which is essentially an agreed upon format for a table in memory that the debugger reads directly and a bit of synchronization around startup, has been widely adopted and is extremely successfully by any measure. Its limitation is that it is predicated on the notion that the set of processes, once established, is static. MPI 2 dynamic processes undermine that assumption.

[5] Reference code and interface header files can be found in the MPICH[1] implementation or can be obtained by contacting Etnus for an up to date version of the interface specification.

2 Design Goals

This interface makes fundamental information about the identity of processes participating in MPI static or dynamic programs available to the debugger so that it can attach to more than just one process of the job. This is being designed within the context of the TotalView parallel debugger, but the challenges of identifying processes in a MPI 2 dynamic program are generally applicable to parallel debuggers as a class and the expectation is that this interface could be adopted by other debuggers. MPI 2 dynamic programs are those that use the dynamic process calls, MPI_COMM_SPAWN, MPI_COMM_CONNECT, MPI_COMM_ACCEPT, and MPI_COMM_JOIN defined in chapter 5 of the MPI 2 standard.[3] However, there are a few other requirements that this interface needs to satisfy.

MPI implementors work to provide the greatest possible performance available and the proposed interface must limit the impact on MPI performance.

The new interface cannot be a step backwards in terms of functionality and needs to support important parallel debugger features like subset attach.

Users need to be able to debug MPI jobs that have deadlocked and hung or that are terminated and exist only as corefiles. This means that the interface needs to allow the debugger to gather the information it needs without running the application or MPI library.

There are wide variety of different resource managers and job launchers in use. It is possible to imagine that TotalView could just interface with them. However we believe this provides no solution for 'singleton' MPI applications and would be needlessly complex for CONNECT / ACCEPT jobs that are started by multiple starters and managed as separate entities by resource managers.

Finally the new interface needs to provide the user with the information that the user will need to make sense of a parallel job. In an MPI 1 job each process is reliably and naturally named by its rank in the communicator MPI_COMM_WORLD. MPI 2 itself does not provide for a global and stable naming scheme from the perspective of the developer who is looking at their code. MPI processes' names for one another are only understood in the context of communicators and communicator handles are purely local. To allow for comparison of results from one run to another with the same input data and program logic, the user will need a way to 'address' their MPI 2 processes in a repeatable way. This paper proposes a stable naming scheme which can be used in debuggers and other tools. The interface will ensure that sufficient information is presented to the debugger for the debugger to construct a meaningful and repeatable name for each process.

3 Design

3.1 Overview

The MPI library itself will maintain a list of processes that are part of the job. As processes are created, CONNECT to, JOIN, or are detached from the program this list of processes will be updated by the MPI library. This list is called the **proctable**. The proctable is distributed across a variable number of MPI processes.

The parallel debugger will be able to read this table from the program using the same mechanisms that it uses to perform other debugging operations.

The MPI library will change the value of a synchronization variable before making changes to the proctable data structures and will reset it afterwards. The MPI library will then call a special stub function to notify the debugger that some dynamic process event caused the proctable to change.

If the debugger is attached to the root process of the dynamic process collective then the newly created MPI processes will be temporarily held to allow the debugger the opportunity to attach to them before the end of MPI_INIT.

3.2 Proctable elements

The proctable contains two kinds of information for each listed process. System context information is needed for the debugger to locate each listed process and potentially be able to attach to it for debugging. MPI context information is needed to uniquely and reliably name each listed process. System information for each process listed in the proctable will include things like: host name or IP address, process or task ID, program name. MPI context information for each process listed in the proctable will include: the rank of each process within its own MPI_COMM_WORLD, a unique identification for that MPI_COMM_WORLD, information about how that MPI_COMM_WORLD came to be part of this job. For the case of MPI_COMM_WORLDs created by a SPAWN operation they can be identified with the following tuple (unique ID of the MPI_COMM_WORLD of the parent root process, rank of parent root process in that MPI_COMM_WORLD, sequence of the SPAWN command among those rooted on that same process). MPI_COMM_WORLDs that are started independently and then connected together need to be given unique ids in this interface that are external to the MPI (e.g. something that is a function of the mac address, PID, and time-stamp of the launcher process).

3.3 Proctable organization

The proctable is distributed across a set of the MPI processes, called the **directory processes**. These processes each contain a subset of the full proctable. A single MPI process may be listed in multiple directory processes; each MPI process is listed in at least one directory process.

In order to reconstruct the proctable the debugger needs to locate all the directory processes and combine their process entry information. Locating the directory processes is done through a set of processes called **meta directory** processes. Meta directory processes each contain a list of directory processes and a list of other meta directory processes. Each directory process must be listed in one meta directory process. The meta directory processes must reference one another such that they form a strongly connected graph. Starting from any meta directory process the debugger must be able to locate the full set of meta directory processes. There can be as few as one meta directory process. Meta directory processes can also be directory processes, in fact is is possible that an implementation would choose to have all directory processes also be meta directory processes. Both directory and meta directory processes are MPI processes.

Meta directory processes are separated in this interface from directory processes because meta directory processes are expected to receive and handle proctable change notification messages from other processes. MPI library implementors may decide that they don't wish to have all the directory processes assume this responsibility. In this case they can have a much smaller set of processes play the role of meta directory processes. Only this narrower set of processes needs to assume the overhead that is involved in processing proctable change messages.

The debugger always needs to be able to identify the meta directory processes. The user should not need to know which process or processes are serving as meta directory processes. So all MPI processes (including all directory processes) will have information about one meta directory process. The user can then connect TotalView to any process of a MPI job and TotalView should be able to discover all the other processes through that one meta directory process.

The intent is that $1 \leq M \leq D \ll R$ where M is the number of meta directory processes, D is the number of directory processes, and R is the number of MPI processes.

3.4 Operations on the distributed proctable

The primary operation that the debugger will need to do on the distributed proctable is list it out. Assuming, for example, that TotalView starts by being attached to any one of the user's MPI process. TotalView then finds the information about the meta directory process that this process references. TotalView then attaches to that meta process and gathers its information. If there are other meta directory processes TotalView walks the graph, attaching to each in turn and gathering a cumulative list of directory processes. Having gathered a full list of directory processes TotalView attaches to any of them that it has not already attached to (remember that meta directories can be directories as well). At this point TotalView is attached to the full list of directory processes and now has the full list of the users MPI processes at hand.

In order to receive notification of proctable changes TotalView will need to remain attached to all of the meta directory processes. If the user does not require notification TotalView can detach from all but one MPI process and still be able to pick up changes to the proctable when the user requests (by reattaching to the **meta directories** locating and attaching to the directories and rereading the proctable information).

MPI library operations on the proctable must be carefully designed to allow for performance at large-scales. For example, modifying the entries in a proctable should involve as few processes as possible – at most, a meta directory process, a directory process, and the processes in the collective action (SPAWN, CONNECT, ACCEPT). It is certainly preferable to involve far less than this (e.g., only the directories and the root process from a SPAWN or representative processes from each of CONNECT and ACCEPT). Since meta directory and directory processes may also be MPI processes involved in the user's application, it is also critical that the interface not require the participating processes to block on the directory process' response.

During normal startup one process might be the meta directory and one or more processes are designated directory process. Information can be propagated as needed to set this up during INIT.

During a SPAWN collective there are two examples that must be considered. If the spawning collective group includes a directory process then that directory gets the information for the newly created processes during the SPAWN collective. A change notice is sent to the meta directory. When the meta directory is able to process the notification it calls a stub routine to notify TotalView that the process table has changed.

If the spawning group does not contain a directory process then one of the group gathers data on its peers and becomes a directory process. This can be done during the SPAWN collective call. The new directory then sends notification to its meta directory process that it has assumed the new status and that a change occurred. The meta directory adds the new directory to its list and calls the stub notification routine for TotalView.

If two separately started jobs are joined with CONNECT and ACCEPT both jobs will have their own preexisting proctable structures. During the CONNECT / ACCEPT collectives the processes participating in the CONNECT / ACCEPT exchange meta directory information. Then one process on each side sends that information (the identity of the other sides meta directory processes) to its 'own' meta directory. When the meta directories each add the new peer to their list of other meta directory processes they make the entire MPI application on the other side of the CONNECT / ACCEPT operation part of the proctable.

CONNECT and ACCEPT can be called within an existing job. If this happens a new connection may be established within the set of meta directory processes but the underlying proctable remains essentially unchanged. Thinking of this as a graph operation, a new pair of vectors are added to an already connected graph but the total set of connected verticies doesn't change.

JOIN operations are almost identical to CONNECT / ACCEPT in terms of operations on the proctable.

3.5 Reading the proctable

The program will expose one or more global loader symbols that TV will use to identify the location in memory to look at to find the information exposed by this interface.

The data will be stored in a structured way that will not depend on the program providing type information to the debugger. It can become complex for

MPI vendors and users to handle the MPI library itself in such a way that the type information is preserved.

One example of an encoding that would meet the above requirement would be a simple pointer to a null terminated string. All the required information could be encoded into this string. Slightly more complex structures of pointers, integers, and strings that have better properties for efficient maintenance will likely be used.

3.6 Synchronization between the MPI and the debugger

During startup the processes will wait for the debugger before proceeding. This can be done using a gate variable in INIT that the debugger has to attach to trigger, or by having a barrier in INIT that the processes all need to reach before running past INIT , or using other mechanisms that achieve the same result.

Synchronization should occur at SPAWN calls if and only if the debugger is attached to the root process of the dynamic process collective. Similar synchronization should occur with CONNECT / ACCEPT , in this case however the newly related MPI processes should be held in the remote collective call, again if and only if the debugger is attached to the root process of the local collective operation.

The MPI library will declare and may check but not set a process level global variable that the debugger can set to notify the MPI process that it is being debugged.

The MPI library will maintain, on a per process level, a variable that the debugger can check to see if the proctable data-structures are being modified.

When the process needs to notify the debugger that an event has occurred it will call an agreed upon stub function. If the debugger wishes to know that this function has been called it can put a hidden breakpoint at that location. This notification will occur on one meta directory when the proctable has changed. It will occur on the root process of a dynamic process collective to notify TotalView that new processes are available and are being held so that the debugger can attach.

4 Naming scheme for MPI 2 proccesses in external tools

A parallel debugger needs a way to identify the many processes being debugged to the user. Each MPI process has a handle to just one MPI_COMM_WORLD, within that MPI_COMM_WORLD each MPI process has a well defind, unique rank. In order to fully and unambiguously specify the process the user needs to have both this rank and a clear way to refer to the MPI_COMM_WORLD. For scripting and comparing the behavior or the program from one run to the next the name that the debugger gives to each MPI_COMM_WORLD should not depend on factors outside the control of the program. In section 3.2 we specified that the proctable will retain information about the MPI process that acted as the root for a newly spawned MPI_COMM_WORLD. This information can be used to construct a name for that new MPI_COMM_WORLD that is unique and descriptive of the specific sequence of SPAWN operations that lead to its creation.

5 Conclusion

The interface discussed here should be useful to any MPI library implementing MPI 2 dynamic processes and any tool designed to work with programs taking advantage of MPI 2 dynamic process features. We will be working first to prototype the MPI library side of the interface in both Open MPI[4] and MPICH 2[5]. At the same time the parallel debugger side of the interface will be prototyped in the TotalView parallel debugger. The design will then be documented based on the experiences and lessons learned in the course of these initial prototyping efforts. This is intended to be an open interface and we welcome input from other MPI and tool developers.

References

1. Gropp, W., Lusk, E.: MPICH. `http://www.mcs.anl.gov/mpi/mpich1/` (2006)
2. Cownie, J., Gropp, W.: A standard interface for debugger access to message queue information in MPI. In: Proceedings, 6th European PVM/MPI Users' Group Meeting. (1999) 51–58
3. Geist, A., Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Saphir, W., Skjellum, T., Snir, M.: MPI-2: Extending the Message-Passing Interface. In: Euro-Par '96 Parallel Processing, Springer Verlag (1996) 128–135
4. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004) 97–104
5. Gropp, W., Lusk, E.: MPICH2. `http://www.mcs.anl.gov/mpi/mpich2/` (2006)